EECE.4810/EECE.5730: Operating Systems

Spring 2018

Programming Project 1

Due **11:59 PM**, **Monday, 2/12/17**

1. Introduction

This project reviews fundamentals of multiprocessing covered at the beginning of the semester. You will write a program that generates multiple processes using the UNIX `fork()` function, handles their return using `wait()`, and generates output from both parent and child processes to demonstrate the correctness of your approach.

This assignment is worth a total of 75 points. The given grading rubric in Section 3 applies to students in both EECE.4810 and EECE.5730.

2. Project Submission

Your submission must meet the following requirements:

- You should submit a single file named `OS_program1.c`.
- Programs must be submitted via e-mail sent to Dr. Geiger (Michael_Geiger@uml.edu)
- You may work in groups of up to 3 on this assignment. If you work in a group, please list the names of all group members in a header comment at the top of your submission, as well as in the email you use to submit your code.

3. Specification and Grading Rubric

As noted above, the program should use `fork()` to start each new process and `wait()` to collect the return status (and possibly check the PID) of each.

Your assignment will be graded according to the rubric on the following page; partial credit may be given if you successfully complete part of a given objective. **You should not submit separate files for each objective, nor should your program contain separate sections or functions for each objective**—your sole submission will be judged on how many of the tasks below it accomplishes successfully. The rubric may also be used as an outline for developing your program—for example, first write a program that accomplishes objective A, then modify it to accomplish objective B, and so on.

In the rubric, objective A is the base case—you cannot complete any of the other objectives without completing that one, and the number of points listed with that objective is essentially the minimum you can earn with a working program. For each additional objective, the number of points will be added to your base score.

Note: it is possible to complete some of the later objectives without completing earlier ones.

Sample outputs from solutions satisfying each objective are shown in Section 5: Test Cases.

3. Specification and Grading Rubric (continued)

Objectives and grading rubric:

A. (15 points) Your program creates a single child process, printing at least one message from both the parent and child process indicating the PIDs of those processes. Your parent process should wait for the child to terminate and print a message once the child has completed.

- The child process should run the same program as the parent process, using a conditional statement to differentiate between the two. That requirement is the same for objectives A-E—having each child process start a new program is required for objectives F & G.
- The messages printed should list the PID of each process.

B. (+11 points) Your program creates multiple child processes without using a loop, printing messages at the start and end of each process as described in part A. **UPDATE: That message should also include a "child number" indicating when each child was created relative to other children (the first child is "Child 1," then "Child 2", and so on.)**

- For this objective and all others that follow, the child processes should run simultaneously, not sequentially. In other words, you should start all child processes, then start using `wait()` to check for child processes finishing. You should not wait for the first child process to finish before starting the second.
- **UPDATE:** As noted above, the "child number" is related to the order in which the child processes are created—not necessarily the order in which they finish. "Child 1" is always the first child process you start, but it may not be the first child that finishes.

C. (+11 points) Your program uses a loop to create ten (10) child processes, printing messages at the start and end of each process as described in part A.

- Completing this objective would give you credit for Objectives A and B—any program that creates multiple child processes satisfies those objectives.

D. (+11 points) Your program is almost identical to part C, but the number of child processes is based on a command line argument passed to your executable, not a constant limit. For example, if your executable is named "`proj1`", executing the command `./proj1 6` will run a version of your program that creates 6 child processes. Assume the maximum number of child processes is 25.

- Completing this objective would give you credit for Objective C—any program that uses a loop to create multiple children, whether the limit is fixed or argument-based, satisfies that objective.

E. (+11 points) Your program is almost identical to part D, but the program is able to discern when each of its child processes completes and print an appropriate message. (For example, when the first child process completes, print a message saying, "`Child 1 (PID` *`xxxxx`*`) finished`", where *`xxxxx`* would be replaced by the actual PID. **The child number and PID printed when each child finishes should match the numbers printed when that child is created.**

4. Specification and Grading Rubric (continued)

F. (+11 points) Your program is almost identical to Part E, but each child process starts a new program, replacing the address space of the parent process with that of the new program. For this part, all child processes should start the same program.

G. (+5 points) Your program is almost identical to Part F, but each child process starts one new program from a set of five possible new programs. Source code for the new programs is on the website. The five test programs are as follows:

1. test1.c: Prints values from 0 to 4, along with the square of each value.
2. test2.c: Calculates and prints the square root of the PID.
   - Since this program uses the math library, you must pass the `-lm` (lowercase L followed by m) option to gcc when compiling test2.c.
3. test3.c: Determines whether the PID is odd or even.
4. test4.c: Calculates and prints the number of digits in the PID.
5. test5.c: Uses a recursive quicksort function to sort an array of ten integers.

UPDATE: Please ensure (1) your executable file names match the .c file names, without the .c (in other words, name them "test1", "test2", etc.) and (2) your program assumes the executables are placed in the same directory as the executable and can therefore be started without listing a full path name. Providing the path "./test1", for example, would specify that "test1" is in the same directory as your executable file.

UPDATE 2: You may NOT modify the source code for the test files. Your program should generate correct output without any changes to those tests.

5. Hints

This section may be expanded in the coming days with more useful information; at this point, my primary goal is to allow you to start the assignment as soon as possible!

Useful functions: The multiprocess examples covered in Lectures 2 and 3 should serve as a starting point for your program. The following additional functions may be useful:

- `pid_t getpid()`: Returns the process ID of the currently running process.
- `int atoi(char *str)`: Converts `str` to the corresponding integer value—for example, `atoi("33") = 33`.
- `int sprintf(char *str, const char *format, … )`: Prints the string specified by `format` and the following arguments to the string `str`. For example, if `x = 7`, `sprintf(s, "x = %d", x)` writes the string `"x = 7"` to the string `s`.

Makefiles: On the Linux machines in Ball 410, you should use the C compiler GCC to compile your code. Repeated compilation is most easily done using the `make` utility, which requires you to write a makefile for your code. Reference material for writing makefiles is easily found online; I found the following website to be a good basic makefile introduction, which is all you should need for this assignment:

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Tracking child processes: In order to ensure that the "child numbers" described in Objective B are generated correctly, you'll have to keep track of the number of child processes created as you do that. Doing so is much simpler if you ensure only one process (most likely the original parent process) can create child processes.

While you can certainly design a solution in which child processes are allowed to create other child processes, that solution may be needlessly complicated.

6. Test Cases

This section provides sample outputs for programs meeting each objective listed in Section 3. Your outputs should match these general forms and must provide all information required for each objective. Your outputs will likely include different PIDs, and statements may be in a different order than these test cases (and from one run of your program to the next).

A. *Single child process:*

```
Parent pid is 4810
Started child with pid 4811
Child (PID 4811) finished
```

B. *Multiple child processes, no loop:* two child processes is the minimum required for this case.

```
Parent pid is 5730
Started child 1 with pid 5731
Started child 2 with pid 5732
Child (PID 5731) finished
Child (PID 5732) finished
```

C. *Ten child processes from a loop:* output will be similar to part B, only with ten statements apiece indicating the start and end of each child.

D. *Number of child processes based on command-line argument:* output will be similar to Part C, with only (major) difference being number of children.

E. *Program can differentiate which child finishes:* example below assumes three children. The main difference between Part E and Parts B-D is each message indicating a child has finished contains both the PID and an identifier ("child 1") indicating the order in which that child was created.

```
Parent pid is 5769
Started child 1 with pid 5770
Started child 2 with pid 5771
Started child 3 with pid 5772
Child 2 (PID 5771) finished
Child 1 (PID 5770) finished
Child 3 (PID 5772) finished
```

5. Test Cases (continued)

F. *All child processes start same executable:* the example below starts 3 copies of "test1". Note that, while each output from that program starts with "T1" to make it clear which test program is running, outputs from the different child processes are interleaved, making it difficult to tell which process is generating each output line.

```
Parent pid is 8556
Started child 1 with pid 8557
Started child 2 with pid 8558
Started child 3 with pid 8559
Running program test1 in process 8558
T1: i 0, i^2 0
T1: i 1, i^2 1
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Running program test1 in process 8557
T1: i 0, i^2 0
T1: i 1, i^2 1
Running program test1 in process 8559
T1: i 2, i^2 4
T1: i 0, i^2 0
T1: i 3, i^2 9
T1: i 1, i^2 1
T1: i 4, i^2 16
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Child 2 (PID 8558) finished
Child 3 (PID 8559) finished
Child 1 (PID 8557) finished
```

G. *Each child process starts one of five executables:* the example below shows a test run with 6 child processes. Note that child 1 and child 6 start the same executable ("test1").

```
Parent pid is 8574
Started child 1 with pid 8575
Started child 2 with pid 8576
Started child 3 with pid 8577
Started child 4 with pid 8578
Running program test1 in process 8575
T1: i 0, i^2 0
T1: i 1, i^2 1
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Started child 5 with pid 8579
Started child 6 with pid 8580
Child 1 (PID 8575) finished
Running program test3 in process 8577
T3: PID 8577 is odd
Running program test2 in process 8576
T2: sqrt of PID 8576 is 92.61
Running program test4 in process 8578
T4: PID 8578 has 4 digits
Child 3 (PID 8577) finished
Child 2 (PID 8576) finished
Child 4 (PID 8578) finished
Running program test1 in process 8580
T1: i 0, i^2 0
T1: i 1, i^2 1
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Child 6 (PID 8580) finished
Running program test5 in process 8579
T5: QS L[0-9]
T5: QS L[0-3]
T5: QS L[0-2]
T5: QS L[5-9]
T5: QS L[5-7]
T5: QS L[5-6]
T5: Final list = 1 2 3 4 5 6 7 8 9 10
Child 5 (PID 8579) finished
```